Introduction tutorial to SPARQL Generate

Introduction

SPARQL Generate is a Domain Specific Language based on SPARQL that allows the translation and homogenisation of heterogeneous data. Through this language the users can merge data sources in different formats in a single RDF Turtle output, creating a simple script. This avoids the creation of *ad-hoc* solutions for every data source and the consequent data homogeniser.

Structure

A script in SPARQL Generate follows the same structure of SPARQL because SPARQL Generate relies in SPARQL for the syntax. Inside a SPARQL Generate script we can find the prefixes declaration, the GENERATE block, the SOURCE clauses, the ITERATOR clause and a WHERE block. Inside a GENERATE block we define the desired shape for the RDF output, the SOURCE clauses allow us to declare the data sources to use, the ITERATOR clauses allow us to define how the algorithm must iterate over the data, and in the WHERE block we can define how the values that we have linked in the GENERATE block are extracted.

Step by step example

In this example we are going to see how a script can be constructed to merge two data sources (JSON and XML) in a single RDF output. The example will be constructed step by step so that in the end we can have the whole view of the created script.

Data sources

The data sources that we are going to use are two: a set of films in JSON and another set of films in XML. The content of these files can be consulted following the links in Table 1. Each file defines two different films with their corresponding attributes: id, name, year, country and directors.

Table 1: Links to the data sources

Format	Link
JSON	https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json
XML	https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.xml

Defining prefixes

To define a prefix that will be used in the script and in the RDF output we use the statement showed in Snippet 1. We use the keyword PREFIX followed by the variable name that we want to use e.g., ex:, schema:, rdfs:, etc. The value for this variable is enclosed between the symbols < and >.

```
PREFIX ex: <http://example.com/>
```

Snippet 1: Declaración prefijo

Defining generators

To define the desired structure that we want for the RDF output we will use the GENERATE block. This block allow us to use the variables that we will define in the WHERE block and then use them to define the structure. This behaviour can be seen in Snippet 2 where ?id_json will be the variable with the expression results and the rest of the variables will generate the objects for each triple. Predicates are declared using the same syntax as in Turtle.

```
GENERATE {
  ?id_json :name ?name_json ;
      :year ?year_json ;
      :country ?country_json .
}
```

Snippet 2: GENERATE block declaration

Defining data sources

To define a data source we will use the SOURCE keyword as it can be seen in Snippet 3. After the keyword the URL to the data source, enclosed between symbols < and >, is declared and, finally, after the keyword AS the variable name to refer to this data source is defined.

```
SOURCE <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json> AS ?films_json
```

Snippet 3: Data source declaration

Defining iterators

Iterators allow us to define when, inside a file, it is necessary to iterate because more than one entity is present. This is the films case in the two previously mentioned files. Therefore, it will be necessary to define in which part of the file the engine must iterate and which elements we will take under consideration. This is achieved in SPARQL Generate with the ITERATOR keyword. This instruction receives an iter function that defines the type of iteration. In the example, we make use of the iter:JSONPath function that receives the JSON file variable and the JSONPath expression to iterate. Finally, the AS keyword is used to assign the iterator result to a variable.

```
ITERATOR iter:JSONPath(?films_json,"$.films[*]") AS ?film_json
```

Snippet 4: Iterator declaration

Defining WHERE block

The WHERE block allows us to define how the variables under the generation block will be extracted and transformed. In Snippet 5 the way how the results for the subjects and objects are extracted from generation block variables is shown. The subject ?id_json is extracted from the function fun:JSONPath that receives two arguments: the iterator variable y the relative query to extract the values. It is worth to highlight that SPARQL Generate assumes that the iterator is a new JSON file, then, the query must be defined from the root. After that it is transformed to a string with the function STR so the subject IRIs can be composed. This string is concatenated with the beginning of the IRI and the it is converted to IRI with the functions CONCAT and IRI respectively. Finally, it is assigned to a variable with the BIND function and the AS keyword. In the case of ?name_json and ?country_json only the extraction function is used because they are going to be literals instead of IRIs. In the case of ?year_json functions CONCAT and IRI are used again to convert them to IRIs. In case a different literal data type was desired (by default xsd:string) the function STRDT can be used using the XML Schema data types as it can be seen in ?name_json. The example can be seen in Snippet 5.

```
WHERE {
  BIND(IRI(CONCAT("http://example.com/", STR(fun:JSONPath(?film_json,"$.id")))) AS
?id_json)
  BIND(STRDT(fun:JSONPath(?film_json, "$.name"), xsd:string) AS ?name_json)
  BIND(IRI(CONCAT("http://example.com/", fun:JSONPath(?film_json, "$.year"))) AS
?year_json)
  BIND(fun:JSONPath(?film_json, "$.country") AS ?country_json)
}
```

Snippet 5: WHERE block declaration

Full example

In the following example we can see how the two previously mentioned files can be merged. In the case of the XML queries to obtain text from the tags the text() function should be used. In addition, in this example it is showed how the different director can be obtained using nested generators. In XML the iter:XPath function is used to make the subiteration and generate various directors. However, to generate the different values in the JSON file the iter:Split function is used because the result can be an array or just a single value. Then, the REPLACE function is used to remove the division remains. In a case where the structure was fixed (i.e., it was always an array) the iter:JSONElement could be used and, then, the iterator values would be accesed with JSONPath query "$.element".

```
BASE <http://example.com/>
PREFIX iter: <http://w3id.org/sparql-generate/iter/>
PREFIX fun: <http://w3id.org/sparql-generate/fn/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://example.com/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>
PREFIX sc: <http://purl.org/science/owl/sciencecommons/>

GENERATE {
 ?id_json :name ?name_json ;
     :year ?year_json ;
     :country ?country_json .

 GENERATE {
  ?id_json :director ?director_json .
 }
 ITERATOR iter:Split(?directors_json, ",") AS ?directors_json_iterator
 WHERE {
  BIND(REPLACE(?directors_json_iterator, "\\[|\\]|\"", "") AS ?director_json)
 } .

  ?id_xml :name ?name_xml ;
     :year ?year_xml ;
     :country ?country_xml .

 GENERATE {
  ?id_xml :director ?director_xml .
 }
 ITERATOR iter:XPath(?film_xml, "/film/directors[*]/director") AS ?directors_xml_iterator
 WHERE {
  BIND(fun:XPath(?directors_xml_iterator, "/director/text()") AS ?director_xml)
 } .

}
SOURCE <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json> AS
?films_json
SOURCE <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.xml> AS
?films_xml
ITERATOR iter:JSONPath(?films_json,"$.films[*]") AS ?film_json
ITERATOR iter:XPath(?films_xml,"//film") AS ?film_xml
WHERE {
 BIND(IRI(CONCAT("http://example.com/", STR(fun:JSONPath(?film_json,"$.id")))) AS
?id_json)
 BIND(STRDT(fun:JSONPath(?film_json, "$.name"), xsd:string) AS ?name_json)
 BIND(IRI(CONCAT("http://example.com/", fun:JSONPath(?film_json, "$.year"))) AS
?year_json)
 BIND(fun:JSONPath(?film_json, "$.country") AS ?country_json)
 BIND(fun:JSONPath(?film_json, "$.director") AS ?directors_json)
 BIND(IRI(CONCAT("http://example.com/", fun:XPath(?film_xml,"/film/@id"))) AS ?id_xml)
 BIND(STRDT(fun:XPath(?film_xml, "/film/name/text()") , xsd:string) AS ?name_xml)
```

```
  BIND(IRI(CONCAT("http://example.com/", fun:XPath(?film_xml, "/film/year/text()"))) AS
?year_xml)
  BIND(fun:XPath(?film_xml, "/film/country/text()") AS ?country_xml)
}
```

Snippet 6: SPARQL Generate script for films

```
@base        <http://example.org/> .
@prefix sc:    <http://purl.org/science/owl/sciencecommons/> .
@prefix schema: <http://schema.org/> .
@prefix :      <http://example.com/> .
@prefix dbr:   <http://dbpedia.org/resource/> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix iter:  <http://w3id.org/sparql-generate/iter/> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix fun:   <http://w3id.org/sparql-generate/fn/> .

:4     :country   "USA" ;
       :director  "Jonathan Nolan" , "Christopher Nolan" ;
       :name      "The Prestige" ;
       :year      :2006 .

:3     :country   "USA" ;
       :director  "Christopher Nolan" ;
       :name      "Inception" ;
       :year      :2010 .

:2     :country   "USA" ;
       :director  "Jonathan Nolan" , "Christopher Nolan" ;
       :name      "Interstellar" ;
       :year      :2014 .

:1     :country   "USA" ;
       :director  "Christopher Nolan" ;
       :name      "Dunkirk" ;
       :year      :2017 .
```

Snippet 7: Turtle result after applying the SPARQL Generate script